A Toolkit for Automated Testing of Dafny

Aleksandr Fedchin[1](✉), Tyler Dean[3], Jeffrey S. Foster[1], Eric Mercer[3],
Zvonimir Rakamarić[2], Giles Reger[2], Neha Rungta[2],
Robin Salkeld[2], Lucas Wagner[2], Cassidy Waldrip[3]

[1] Tufts University, Medford, USA
{aleksandr.fedchin,jeffrey.foster}@tufts.edu
[2] Amazon Web Services, Seattle, USA
{zvorak,reggiles,rungta,salkeldr,lgwagner}@amazon.com
[3] Brigham Young University, Provo, USA
egm@cs.byu.edu, {cassidy.waldrip,tylerdean779}@gmail.com

Abstract. Dafny is a verification-ready programming language that is executed via compilation to C# and other mainstream languages. We introduce a toolkit for automated testing of Dafny programs, consisting of DUnit (unit testing framework), DMock (mocking framework), and DTest (automated test generation). The main component of the toolkit, DTest, repurposes the Dafny verifier to automatically generate DUnit test cases that achieve desired coverage. It supports verification-specific language features, such as pre- and postconditions, and leverages them for mocking with DMock. We evaluate the new toolkit in two ways. First, we use two open-source Dafny projects to demonstrate that DTest can generate unit tests with branch coverage that is comparable to the expectations developers set for manually written tests. Second, we show that a greedy approach to test generation often produces a number of tests close to the theoretical minimum for the given coverage criterion.

1 Introduction

Verification-ready languages and tools, such as Dafny [12,21,22] and Boogie [3], extend a core programming language with support for formal specifications such as preconditions, postconditions, and loop invariants. Developers verify programs against such specifications using built-in verifiers, thereby reducing the risk of hidden bugs. Verification-ready languages have been successfully used in scenarios ranging from low-level hypervisors [20] to entire program stacks [16].

It is common for a program written in a verification-ready language, such as Dafny, to first be compiled into a traditional programming language, such as C#, before being deployed to production. This way one can leverage the extensive compiler optimizations and libraries that have already been developed for popular programming languages. At the same time, one also needs to guarantee the correctness of the final deployed program. First, it is necessary to ensure that the Dafny compilers, such as the Dafny to C# compiler, do not introduce unexpected behavior [18]. One approach to ensure the correctness of a compiler

would be to verify it end-to-end. There have been such efforts in the past for other languages [23]. While successful, these efforts took years of manual human effort, and Dafny supports compilation to several different languages making verification of the entire toolchain very difficult. Second, many Dafny programs use external libraries, which are another potential source of bugs since they are not written in Dafny and hence are not verified to match their specifications. Incorrect specification of an external library may introduce bugs even if the library itself and the entire compilation pipeline are verified to be correct.

In this paper, we propose to increase assurance of the correctness of the compiled Dafny program by leveraging automated testing. More specifically, we introduce a toolkit for automated testing of Dafny programs, consisting of DUnit (unit testing framework), DMock (mocking framework), and DTest (automated test generation). The main purpose of the combined toolkit is to ensure that the guarantees provided by verified Dafny programs are preserved when those programs are executed via compilation to a different programming language, such as C#. The main component of the toolkit is DTest, a tool for automated generation of tests that achieve high coverage of Dafny programs. The tests themselves are written in Dafny and compiled to use testing frameworks in selected target languages, including C#. The tests assert that method postconditions verified in Dafny hold at runtime. Thus, we can use DTest to (i) generate tests to ensure a compiled program preserves the behavior verified in Dafny; (ii) increase confidence in specifications of external libraries that cannot be verified; and (iii) increase assurance that a Dafny program is functionally equivalent to an existing implementation that may be written in another language.

To compile tests to the target programming language, we introduce DUnit and DMock, unit testing and mocking frameworks for Dafny. DUnit extends Dafny with a method attribute :test, which signals the compiler to mark the corresponding method as a unit test in the testing framework of the target language. To support DUnit, DMock facilitates generation of complex heap structures as test inputs by adding mocking capabilities to Dafny. We introduce a new Dafny attribute (:synthesize) for tagging of mock methods, which have no body in Dafny but instead describe their return values with postconditions. Tests produced by DTest rely on mock methods to bypass the need to infer how to use existing constructors to create objects with specific field values. Instead, DMock automatically compiles mock methods to code (using the popular Moq mocking framework for C#) that returns objects that comply with the corresponding postconditions. Currently, DMock can produce mock implementations for a specific but broadly useful set of postconditions: one can supply concrete values for constant instance fields or redefine the behavior of instance functions.

Fig. 1 shows the typical toolflow of the Dafny testing toolkit. DTest is implemented as an extension to Dafny and uses the existing Dafny verifier, which works by translating the Dafny program to the Boogie intermediate verification language [3, 7]. Boogie, in turn, proves each assertion with Z3 [29, 33]. DTest starts test generation by translating Dafny to Boogie (step 1 in the figure), including several changes to the existing translation pipeline (see Sec. 4.1). Next,

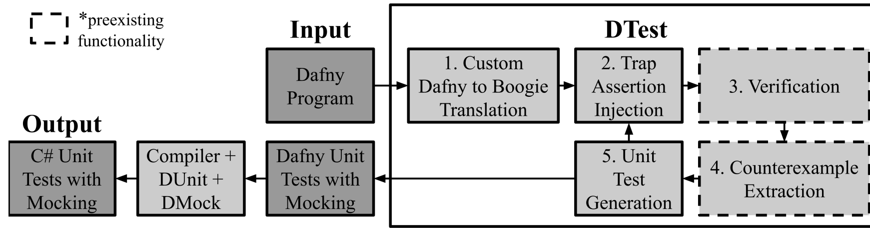

Fig. 1: Toolflow of the Dafny testing toolkit.

DTest enters a loop where it systematically injects trivially failing *trap assertions* (meaning `assert false`) into the Boogie code and uses the Boogie verifier and counterexample extractor [8] to generate counterexamples that reach the assertions (steps 2–4). Then, DTest translates counterexamples into Dafny tests (step 5) using unit testing and mocking attributes understood by DUnit and DMock, and converts method postconditions into runtime oracles (see Sec. 4.3). We then compile the Dafny program and the generated tests to C# using the Dafny compiler augmented with the functionality that DUnit and DMock provide.

We evaluated our toolkit across two dimensions. First, we used DTest to generate unit tests for the Dafny utilities library (DUTIL) [24] and the portion of the AWS Encryption SDK (ESDK) that is implemented in Dafny [13]. We then compiled each library and its tests to C# and measured the coverage of the tests on the C# code: the tests produced by DTest achieved 79% (resp. 62%) statement and 84% (resp. 58%) branch coverage on DUTIL (resp. ESDK). This is promising since the ESDK developers target 80% statement and 35% branch coverage for their manually written unit tests as part of their wider testing strategy. Second, we compared the number of tests DTest generates to achieve full coverage to the number of tests generated by a brute-force algorithm that can optimally minimize the number of tests. We found that DTest often generates close-to-the-minimal number of tests, with the worst observed case (for some of the methods with the most complex control flow) being three times the optimal.

In summary, the main contributions of this work are as follows:

– We introduce DTest, a tool that uses the Dafny verifier to automatically generate unit tests for preexisting Dafny programs.
– We develop DUnit and DMock, unit testing and mocking frameworks for Dafny that support automated compilation of tests and construction of objects based on a formal description of their behavior.
– We evaluate the toolkit on a set of real-world Dafny programs and show that the generated tests achieve coverage expected by the developers.
– We released our toolkit with Dafny [12] and made the persistent artifact for the paper available at `https://doi.org/10.5281/zenodo.7310719`.

Overall, our results show that DUnit, DMock, and DTest are a promising toolkit for automatically generating high coverage tests for Dafny. More broadly, our work should be useful to researchers and practitioners working in verification-ready ecosystems other than Dafny, as we provide solutions for critical pain

points in test generation, including dealing with pre- and postconditions, mocking in their presence, and leveraging the verifier for automatic test generation.

2 Toolkit Overview

Fig. 2a gives the example Dafny method LexLeq (LexicographicByteSeqBelowAux originally) we extracted from the ESDK to illustrate how DUnit and DTest work. It takes byte sequences x and y and an index n as input, and returns a Boolean indicating whether x is equal to or precedes y in lexicographic order starting at position n. The core logic of the method (lines 5–7) is a disjunction of conditions that would make this true: either we have reached the end of x, or the byte at position n in x comes before y, or the two bytes are equal and x is before y lexicographically at position n+1. Otherwise, x is greater than y at n. Because the method is recursive, it is accompanied by a decreases clause (line 4), which allows Dafny to prove termination by stating that at each recursive call the value of $|x| - n$ decreases. The method also has a precondition (line 2) requiring that n is within a valid range for x and y, and a postcondition (line 3) ensuring that if the result is true then either we have reached the end of x or we have not reached the end of y. Note that the postcondition was not present in the original code, but we added it to more fully illustrate the features of DTest.

Dafny verifies programs by translating them to the Boogie intermediate verification language [3, 7] and then verifying the Boogie code. For our example, DTest translates the Dafny code in Fig. 2a to the Boogie implementation in Fig. 2b. Note that this translation differs from one the regular Dafny to Boogie translator would produce—we discuss the differences in Sec. 4.1. The code in Fig. 2b takes three input parameters that directly map to the parameters in the Dafny code. The parameters x and y have type Seq Box, which is the type that the Dafny translator uses to encode sequences in Boogie. For clarity, we use Dafny notation in place of Boogie function calls for element access, a[i], and sequence length, $|x|$. On entry to the implementation, the Boogie program proceeds to either block A or B, each corresponding to one of the two possible values of the Boolean expression on line 5. Note that in Boogie, control flow is captured by non-deterministic branches to blocks guarded by assume statements. For example, here, block A is guarded by an assumption $n \neq |x|$ falsifying the condition on line 5 in Fig. 2a. Thus, the Boogie code has a block for each term of the Boolean expression in the original Dafny. Therefore, block coverage of the Boogie code essentially corresponds to branch coverage of the original Dafny code.

Recall that DTest finds inputs that reach target branches by iteratively inserting assert false in each block and then extracting a counterexample from the verifier. We call such assertions *trap assertions* because we do not expect the prover to successfully verify them. Here, DTest has added a trap assertion on line 19 in the Boogie code, with the goal of covering block L.

When we ask Boogie to verify the code in Fig. 2b, the verifier produces a counterexample. The counterexample itself is not human readable, but recent work [8] allows us to infer counterexample arguments x=$[0, 133, 188]$, y=$[0, 133, 187]$, and

```
1  function LexLeq (x: seq⟨uint8⟩, y: seq⟨uint8⟩, n: nat): (result: bool)
2    requires n ≤ |x| ∧ n ≤ |y|
3    ensures result ⇒ n ≡ |x| ∨ n ≠ |y|
4    decreases |x| − n {
5       n ≡ |x|
6    ∨ (n ≠ |y| ∧ x[n] < y[n])
7    ∨ (n ≠ |y| ∧ x[n] ≡ y[n] ∧ LexLeq(x, y, n + 1))}
```

(a) An example Dafny method from the ESDK.

```
8  implementation LexLeq (x: Seq Box, y: Seq Box, n: int)
9    returns (result: bool) {
10   var tmp: bool;
11   Entry: goto A, B;
12   A: assume n ≠ |x|;
13      goto C, D;
14   B: assume n ≡ |x|;
15      goto E;
16   // blocks C to K go here...
17   L: assume n ≠ |y| ∧ x[n] ≡ y[n];
18     call tmp := LexLeq(x, y, n + 1);
19     assert false; // Added by DTest
20     goto Return;
21   Return:
22     result := n ≡ |x| ∨ (n ≠ |y| ∧ x[n] < y[n])
23                       ∨ (n ≠ |y| ∧ x[n] ≡ y[n] ∧ tmp);
24     return;}
```

(b) (Simplified) Boogie translation with a trap assertion.

```
25 method {:test} test() {
26   var d0: seq⟨int⟩ := [0, 133, 188];
27   var d1: seq⟨int⟩ := [0, 133, 187];
28   var r0 := LexLeq(d0, d1, 1);
29   expect r0 ⇒ 1 ≡ |d0| ∨ 1 ≠ |d1|;}
```

(c) Generated Dafny unit test.

```
30 [Xunit.Fact]
31 public static void test() {
32   var d0 = Sequence⟨int⟩
33     .From(0, 133, 188);
34   var d1 = Sequence⟨int⟩
35     .From(0, 133, 187);
36   bool r0 = LexLeq(d0, d1, 1);
37   if (r0 ∧
38      d0.Count ≠ 1 ∧
39      d1.Count ≡ 1)
40     throw new Dafny.Exception();}
```

(d) (Simplified) generated C# unit test.

Fig. 2: Unit test generation example.

n=1. DTest then uses these arguments to produce the unit test in Fig. 2c. Here the :test annotation signals to the compiler that this method should be compiled as a unit test in the target language of choice. The body of the test begins by constructing sequences d1 and d2 which, along with the literal 1, are the counterexample arguments. Next, the test case calls LexLeq with these arguments. The expect statement on line 29 is a runtime assertion. Here we check that the result satisfies the postcondition. Thus, the test not only covers block L, but also adds a level of assurance to the emitted code by introducing runtime checks.

Finally, Fig. 2d shows the C# unit test DUnit generates for the example Dafny unit test. Lines 32–35 correspond to lines 26–27 in Dafny and construct the counterexample arguments later used in the method call on line 36. The conditional on lines 37–40 throws an exception in case of a postcondition violation. Note that DUnit converts the :test annotation in Dafny to XUnit.Fact, which allows us to run the resulting test using .NET's XUnit framework [32].

3 Unit Testing and Mocking Frameworks

To support DTest, over the span of several years we developed DUnit and DMock, unit testing and mocking frameworks for Dafny. In this section, we describe the new unit testing and mocking constructs we introduced to the Dafny language as well as how we compile them into C#, the target compilation language used by most open-source Dafny projects.

As we discuss in Sec. 2 (see Fig. 2c), DUnit introduces the :test attribute for annotating unit tests. Within a unit test, we introduce expect statements to specify runtime assertion checks. In contrast to standard Dafny assert statements, the Dafny verifier does not prove expect statements but instead assumes they hold. Dafny compiles expect statements into runtime assertions in the target language, whereas assert statements are removed from compilation.

In addition to this basic unit testing support, we also introduce support for runtime *mocking*, which allows seamless creation of objects based on a description of their behavior. When compiling to C#, we translate Dafny mocks into code that uses the popular Moq library [28]. (Note that DMock also supports compilation to Java using the Mockito library [27], but we focus on C#.)

The key reason we developed DMock is to support the creation of heap-based structures (i.e., objects), which DTest heavily relies on. In particular, mocking solves the problem of having to synthesize a sequence of calls to constructors and other API methods to put a given object into the required state. In DMock, we introduce the :synthesize attribute for annotating mock methods, which is accompanied by postconditions describing the method's return value. We can use such postconditions to specify mocking behavior of constant instance fields and functions. DTest can infer from counterexamples the arguments with which to call mock methods, and we do not allow mocking of side-affecting properties, which ensures that the objects are consistent with preconditions and type invariants.

Fig. 3a gives an example mock method that generates a new AwsKmsKeyring object and sets its instance fields to values given to the method call as arguments.

```
1  method {:synthesize} getKeyring
2    (client: ManagementServiceClient, key: String, arn: Identifier, tok: TokenList)
3    returns (o: AwsKmsKeyring)
4    ensures fresh(o)
5    ensures o.client≡client ∧ o.Key≡key ∧ o.Arn≡arn ∧ o.Tokens≡tok
```

(a) Dafny.

```
6  public static AwsKmsKeyring getKeyring
7    (ManagementServiceClient client, String key, Identifier arn, TokenList tok) {
8    var mock = new Mock⟨AwsKmsKeyring⟩();
9    mock.CallBase = true;
10   mock.SetupGet(x ⇒ x.client).Returns(client);
11   mock.SetupGet(x ⇒ x.Key).Returns(key);
12   mock.SetupGet(x ⇒ x.Arn).Returns(arn);
13   mock.SetupGet(x ⇒ x.Tokens).Returns(tok);
14   return mock.Object;}
```

(b) C#.

Fig. 3: An example use of the :synthesize attribute for mocking (simplified).

DTest automatically produced this method (simplified) and relevant arguments to call it with while generating tests for the ESDK. First, line 4 uses Dafny's fresh keyword to ensure the returned object is new and not aliased by an existing variable. The subsequent postconditions specify the values of each of the object's constant fields. DMock compiles this method into the code in Fig. 3b. On line 8, we use Moq to create a new class that extends AwsKmsKeyring. On the next line, we ensure that by default the mocked class behaves exactly as the original class it extends. Then, we override the field getter methods to return the values provided as arguments. Finally, we return a new instance of the mocked class that, by construction, behaves exactly as specified by the postconditions in Fig. 3a.

DMock also supports mocking of instance functions by redefining their behavior with arbitrary expressions. For example, we can add the following postcondition to the mock method in Fig. 3a to ensure that a call to the Identity instance function simply returns its argument: `ensures forall arg:int :: o.Identity(arg) ≡ arg`. DMock compiles this postcondition into the C# statement below (which would be added to Fig. 3b) to override the behavior of the Identity function:[1]

```
mock.Setup(x ⇒ x.Identity(It.IsAny⟨BigInteger⟩())).Returns((BigInteger arg) ⇒ arg);
```

This functionality is particularly useful for instantiating traits, which are Dafny types similar to interfaces in Java that also cannot be instantiated directly. A method annotated with :synthesize can both return an object extending a given trait and ensure the instance functions of that object behave as the postconditions dictate. Note that we can only mock instance functions, not methods,

[1] Dafny's int is compiled to C#'s BigInteger because in Dafny integers are unbounded.

since method calls cannot appear inside postconditions, which are expressions. However, Dafny programs are typically written in a functional style, and hence DTest can still handle most real-world uses of traits.

4 Automated Test Generation

In this section, we describe DTest's test generation approach (steps 1-5 in Fig. 1).

4.1 Custom Dafny to Boogie Translation

DTest customizes (step 1 in Fig. 1) Dafny's standard translation to Boogie with two key modifications to support automated test case generation.

Preprocessing to Support Inserting Trap Assertions. The Dafny code we have analyzed makes extensive use of functions. Function bodies are syntactically expressions and are translated as such into Boogie. However, an assertion is a statement, and cannot be inserted into the body of a Boogie function. To address this issue, DTest preprocesses the Dafny code to turn functions into function-by-methods, which are functions with an equivalent imperative definition provided as a method. In our case, we wrap the original expression in a return statement, which then prompts the translator to create an imperative Boogie implementation. Hence, for each input function-by-method, Dafny emits both a standard Boogie function—used for verification—and an imperative implementation, as in Fig. 2b. DTest can then insert trap assertions into implementations' bodies.

Inlining. If we are using DTest to generate unit tests of individual methods, no further translation steps are needed. However, an issue arises if we wish to generate system-level tests via calls to a `main` method entry point or similar. The challenge is that Boogie verifies methods one at a time, and any callee methods are represented by their specifications. Any trap assertions aside from those in a `main` method will essentially be "hidden" behind the specifications of the methods they are inside of.

Our solution to this problem is inlining: DTest can optionally inline the program into a user-specified `main` method before proceeding with test generation. Recursive methods can also be inlined (unrolled) up to a manually chosen bound. This way, DTest can provide coverage of the entire Dafny program. Boogie supports inlining, but to take advantage of this support, we have made several changes to the Boogie code emitted by Dafny. These changes allow translating functional-style code, such as conditional expressions, to their imperative equivalents, such as conditional statements, which makes the code more amenable to trap-assertion injection and inlining.

4.2 Trap Assertion Injection

DTest generates tests while iterating over the basic blocks of the Boogie representation. Iteration happens in reverse topological order of the control-flow

graph in order to greedily reduce the number of tests by generating tests that are likely to cover multiple blocks at a time. For each block that has not yet been covered, DTest inserts a trap assertion (step 2 in Fig. 1) and queries the Boogie verifier for a counterexample (step 3). Alongside the counterexample, the verifier also reports the error trace, i.e., blocks leading up to the trap assertion that the counterexample also covers. We use the error trace to prune away already covered blocks. DTest then uses previous work [8] to extract the counterexample to a Dafny-like format and then concretizes the result (step 4 in Fig. 1).

One can construct more complex trap assertions that fail when a program takes a specific path through the control flow graph. We can, therefore, use DTest to generate test suites with path-coverage guarantees, although we do not fully explore this use case here and only apply this version of DTest to study the sizes of potential test suites (see Sec. 5.2.)

Note that a successfully verified trap assertion serves as proof that no input can cause a given block to be visited, i.e. it signifies the presence of dead code. This also allows us to uncover dead code using DTest, which is an option we implemented but have not experimented with extensively.

4.3 Unit Test Generation

The key challenge DTest faces when generating unit tests (step 5 in Fig. 1) involves selecting concrete values that are not constrained by the counterexample because they are irrelevant to a particular assertion failure. For example, consider the method in Fig. 2a. A counterexample returned by the solver may suggest that calling the method with $n = 1$ and x being a one-element sequence covers block B. To generate a unit test, DTest also has to emit a value for x's single element. DTest is free to choose any value assuming it satisfies the corresponding type constraint if any such constraint is present.

To generate such values, DTest relies on *witnesses*—user-supplied (using the `witness` keyword in Dafny) or sometimes automatically inferred values that Dafny uses to prove a given type is nonempty. We define such values for all primitive types and collections (e.g., `0` for integers), and user-defined witnesses are typically available for subset types, i.e., types that are constrained with arbitrary predicates. In the rare case that a user does not suggest a witness for a given subset type, DTest will emit a default value for the corresponding supertype, which may lead to a test that violates a type constraint. We call any such test that violates the specification of the target method or a type constraint *unreliable* and discuss all cases in which DTest might generate such tests in Sec. 4.4.

One way to exclude unreliable tests would be to verify them in Dafny. However, DTest might generate a correct test while at the same time failing to find the right value for a ghost variable — irrelevant at execution — to make the test verify. Moreover, some tests may be unreliable yet still explore the targeted branch when compiled to C#. Therefore, to allow more flexibility, we aim to filter unreliable tests at runtime with checks that preemptively terminate execution if a test violates a method precondition over non-ghost fields, violates a type constraint, or calls a trait instance method that is not explicitly mocked

by DTest. For example, in Fig. 2c we add such a check after the last local variable initialization on line 27 as the runtime assertion `expect` $1 \leq |\mathsf{d0}| \land 1 \leq |\mathsf{d1}|$, `"Unmet precondition"`. In our evaluation, such runtime checks terminate 94% of all unreliable tests that would otherwise have led to difficult-to-interpret failures.

To strengthen the assurance provided by the tests, DTest converts postconditions of methods under test into runtime assertions such as the one on line 29 in Fig. 2c. We support all specification constructs allowed by Dafny, provided they are not ghost. We leave compilation of ghost constructs such as unbounded quantifiers as future work. Whenever DTest encounters ghost specifications, which are infrequent in our evaluation, it does not create a corresponding runtime check; this, of course, does not affect coverage.

4.4 Limitations

DTest has several limitations that can either prevent it from being able to generate a test for every block in the Boogie representation of a given procedure or might cause DTest to occasionally produce unreliable tests (Sec. 4.3).

- **Solver Timeouts.** As is the case with any tool that relies on an SMT solver, timeouts may occur, in which case DTest might not cover some blocks. We currently set the timeout to 5 seconds, and our empirical evaluation shows that increasing the timeout does not make a significant difference.
- **Spurious Counterexamples.** Dafny might generate a spurious counterexample (i.e., one that does not in fact lead to a trap assertion violation) due to several reasons. First, specifications, such as post-conditions or loop invariants, might be over-approximations that under-constrain the program state. Second, the Dafny translation into Boogie might not provide a complete axiomatization of some features, such as set cardinality. Third, the backend SMT solver itself is incomplete in the presence of quantified formulas, which Dafny always generates. This can lead to a counterexample that does not expose a trap assertion or may even violate method preconditions.
- **Information Elided in the Counterexample.** If the user does not provide a witness for a certain subset type (Sec. 4.3), DTest may not be able to generate a value that satisfies the corresponding type constraint.
- **Ghost Specifications.** DTest cannot compile ghost specifications into runtime checks, so there could be unreliable tests we fail to identify.
- **Unsupported Language Features.** DTest does not support tuples, arrays, infinite maps, infinite sets, or multisets. These Dafny types and collections are rarely used in practice (sequences are used instead of arrays; finite maps and sets are preferred to their infinite counterparts) and only appear in a handful of methods in our benchmarks. Moreover, DTest does not fully support traits and function types. For any argument of a function type, DTest synthesizes a lambda expression with a matching type signature. For example, for a function that maps an integer to an integer, DTest synthesizes a lambda expression that always returns `0`. Given that function types are rarely constrained, this approach works in the majority of cases. For a discussion of traits, see Sec. 3.

5 Empirical Evaluation

We perform two experiments to evaluate the testing toolkit. First, we evaluate DTest's running time and coverage achieved on two preexisting Dafny projects. Second, we compare the number of tests DTest generates to the minimal number of tests required for full coverage.

The subject programs for our evaluation are two real-world projects: the Dafny utilities library (DUTIL) [24] and the AWS Encryption SDK (ESDK) [13]. DUTIL spans 1382 lines of code and presents a collection of useful methods for non-linear arithmetic; manipulating Dafny maps, sequences, and sets; and performing miscellaneous operations. The ESDK comprises 4596 lines of code and implements a Dafny-verified encryption library, which provides an interface between encryption backends and consumer applications. The two projects are, to the best of our knowledge, the largest open-source Dafny programs, with the exception of Ironclad [16] which, despite manually updating it to the latest Dafny syntax, we were unable to get to verify (and hence use in our experiments). While our benchmarks are small by industry standards, they are representative of how Dafny is used in practice and showcase most of Dafny's features.

5.1 Unit Testing and Coverage

In our first experiment, we measure the statement and branch coverage on the binary obtained by compiling DUTIL and the ESDK to C#. To maximize DTest's performance, we augmented the ESDK with about two dozen *witnesses* (Sec. 4.3). Doing so took us less than an hour of manual work. We find that DTest can quickly generate tests that provide sufficient coverage to identify unexpected behavior in an external library.

Performance. DTest took 158 minutes to generate 918 tests for the 436 methods in the two benchmarks. This does not include methods that exist only to aid verification and are not compiled, methods that have no body in Dafny (external methods), and methods introduced by the Dafny compiler (e.g., ToString). Fig. 4 shows that the runtime it takes DTest to process one method is close to linear in the number of blocks in the Boogie representation of that method (Pearson's coefficient ≈ 0.86, $p < 0.0005$). The outliers are methods for which DTest can

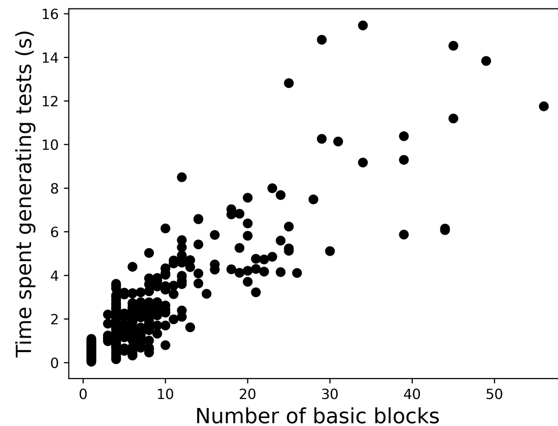

Fig. 4: Runtime of DTest on methods from ESDK and DUTIL.

Table 1: Overview of achieved coverage.

	Source Folder	LOC	Methods	# Tests	Boogie Block	C# Statement	C# Branch
DUTIL	Maps	55	7	13	100%	98%	88%
DUTIL	Sequences	1114	131	169	67%	76%	84%
DUTIL	Sets	42	4	8	100%	83%	69%
DUTIL	Nonlinear Arithmetic	78	7	11	72%	91%	88%
DUTIL	Misc.	93	16	22	90%	98%	88%
	Total	**1382**	**165**	**223**	**72%**	**79%**	**84%**
ESDK	Crypto Material Providers	1871	67	270	87%	62%	50%
ESDK	Crypto	153	13	25	90%	82%	70%
ESDK	Generated	61	4	8	83%	100%	100%
ESDK	SDK	1878	116	205	58%	51%	45%
ESDK	Standard Library	414	45	121	90%	93%	93%
ESDK	Util	219	26	66	74%	75%	91%
	Total	**4596**	**271**	**695**	**77%**	**62%**	**58%**

cover multiple blocks with one test, such as the methods we analyze in Sec. 5.2. Methods that cause solver timeouts are also a source of outliers.

Coverage. Table 1 shows the coverage that the tests achieve on the compiled C# code, as measured with the Coverlet framework [11]. We give the results for each source folder in DUTIL and ESDK, including the lines of code (LOC) count, target method count, and the number of generated tests for each entry. Each folder consolidates files with similar functionality. The table includes a *Block* column showing the fraction of Boogie basic blocks for which DTest reports that it successfully generated tests. We report the actual statement and branch coverage in the last two columns, with unreliable tests (Sec. 4.4) not contributing to the result. We took all methods compiled from Dafny to C#, even those for which DTest fails to generate tests, into account when measuring coverage. We achieve 62% statement and 58% branch coverage on the ESDK. The lower coverage of the SDK folder is due to extensive use of traits, which DTest only partially supports (Sec. 4.4). These results are comparable to the thresholds that the ESDK developers set as a minimum bar for manual tests as part of their overall testing and verification strategy, with DTest scoring above the 35% threshold for branch coverage but below the 80% threshold for statement coverage.

We observe in the table that the Boogie basic block coverage does not match the C# statement coverage, but is either an over- or under-approximation. The difference is due to two key factors. First, a C# test generated for one method might cover code in a method invoked by it, which at the Boogie level we cannot observe since we are doing intraprocedural test generation in the experiments (no inlining). This leads to the Boogie coverage being an under-approximation of the C# coverage. Second, some tests may be identified as unreliable at runtime, and so they contribute to the Boogie coverage but not to the C# coverage, leading to the former being an over-approximation of the latter.

Overview of Tests. Of the 918 tests DTest generated for DUTIL and ESDK, 85 are unreliable. Of these, our runtime checks preemptively terminated 80 (see Secs. 4.3 and 4.4): 11 violate preconditions, 40 violate type constraints, and 29 call methods on traits that are not mocked. Five more tests fail because DTest does not fully support function types. As we describe above (Sec. 4.4), arguments of function types are generated from type signatures rather than counterexamples. Overall, half of the generated tests have no return value check due to absence of postconditions. However, even such tests check for runtime errors, which is valuable as shown by the success of black-box fuzzers (that do not check return values either) [26]. In our experiments, the outcomes of two tests are worth noting: one test causes an external method to throw an exception, which, however, is allowed by that method's signature. Another test causes the execution to continue for an indefinite amount of time (we killed the process after two hours). The developers identified this case to be from a particular internal test method (not part of code exposed to the user) for an external .NET RSA library. Thus, our test uncovered an external library behaving differently from the developers' expectations, and they fixed the test method accordingly.

5.2 Test Suite Size

In our second experiment, we evaluate the size of the test suite DTest generates by comparing it to a minimal number of tests required to achieve full coverage. Test suite size is an important factor for software development and has prompted significant research effort in recent years [15, 17].

For the purpose of this comparison, we designed an algorithm to enumerate sets of control flow paths of a Boogie procedure in order of increasing set cardinality, terminating when we find a set of paths that guarantee full coverage, are all feasible, and that we can generate tests for. We determine the feasibility of a path via a query to the SMT solver, in which a trap assertion is added that fails only if all the blocks along the path are visited. We then generate a test for each path in the same way that DTest would generate a test for a given block. This approach is exponential in the number of SMT queries (running on all benchmarks as in Table 1 would take weeks), but we do allow the users of DTest to optionally use this costly method since reducing the number of tests is sometimes of utmost importance (e.g., if tests are being executed over and over again as a part of continuous integration).

To compare the default (greedy) and optimal approaches, we selected nine methods from DUTIL and ESDK with the most complex control flow as determined by the number of basic blocks in their Boogie representation. Such methods present high potential for test minimization, since one carefully chosen test could cover many blocks; we expect differences between the two approaches to be less pronounced when viewed in a broader context. We omit any methods for which DTest could not reach all blocks, or for which it generated unreliable tests.

Table 2 summarizes the results of this comparison. We give the number of basic blocks in the Boogie representation of each method in a separate column.

Table 2: Comparison of minimization strategies.

Method	# blocks	Running time (s)		# Tests	
		Default	Optimal	Default	Optimal
UTF8.Uses3Bytes	53	18	1860	13	12
UTF8.Uses4Bytes	53	19	422	13	12
UTF8.ValidUTF8Range	21	12	36	9	6
Base64.IsBase64Char	19	16	112	8	3
AwsKmsArnParsing.AwsArn.Valid	16	17	30	6	2
Base64.IndexToChar	16	14	19	5	5
Base64.Is1Padding	16	13	24	6	2
Base64.Is2Padding	16	14	25	6	2
Sorting.LexicographicByteSeqBelowAux	14	8	8	5	3

For each Dafny method and minimization technique, we report the time that the test generation process takes (mean of three runs), and the number of tests in the resulting suite. The default (greedy) algorithm appears to generate more tests whenever there are independent branching points in the control flow of the method, i.e., when the choice of the path at one branching point does not dictate the choice at the next one. Even so, the greedy algorithm is within two tests of minimal for four of the methods, and it never generates more than three times the minimal number of tests. It is also several times faster than the alternative.

6 Related Work

Testing of Verification-Ready Languages. Test generation has been explored in the context of Dafny by Delfy [9, 31], a concolic test generation tool. Delfy has not been updated for nearly ten years and only supports a limited subset of Dafny, and hence a direct comparison was not possible. In contrast to Delfy, DTest fully supports the features commonly used by Dafny programmers, such as algebraic datatypes, sequences, sets, and maps. Unlike Delfy, which relies heavily on compilation to C# for both concrete and symbolic execution, DTest is independent of the target language since it generates tests (in Dafny) from counterexamples provided by the Dafny verifier itself.

Another tool for automated testing of Dafny is XDSmith [18], which randomly generates Dafny programs with known verification outcomes and uses these programs to test the Dafny verifier and compilers. XDSmith is complementary to DTest—the former focuses on testing the Dafny toolchain itself in isolation, while the latter helps to increase assurance that the compiled target programs are correct, particularly in their interaction with external libraries.

Concrete execution of verifier-produced counterexamples has been explored in the Why3 verification environment [4,14]. The goal of this work is to ascertain the validity of a counterexample by observing the runtime behavior it triggers under various assumptions. DTest, by contrast, relies on the correctness of counterexamples to generate tests. This technique and, more broadly, the use of a verifier to generate tests has been explored in the context of other languages,

such as C [5] and B [1], but Dafny presents a particular challenge due to its verification-related features (e.g., rich type system, specifications).

To the best of our knowledge, neither Boogaloo [30], a nondeterministic interpreter for Boogie, nor Symbooglix [25], a symbolic execution engine, have been used to generate test suites for Boogie or Dafny programs although both tools can be used to explain a failing assertion. Symbolic-execution-based program exploration algorithms and the verifier have different trade-offs, especially in the presence of loops. We plan to develop, as future work, a portfolio-based approach, similar to CoVeriTest [6], with several backend reachability analyses.

Automated Software Testing. There is a large body of work on automated software testing, involving techniques such as fuzzing (see [26] for a survey), symbolic execution [2], and others. We might augment DTest with some of these techniques in the future since, for example, the approach used by QuickCheck [10] and the related family of fuzzers for generating values of function types offers more flexibility than DTest's current implementation. One of the challenges often accompanying automated testing is object initialization. Our approach to this problem is close to lazy symbolic initialization [19], a process whereby an object is initialized on an "as-needed" basis—we similarly override the value of an object's field only if it is constrained by the counterexample or a precondition.

Mocking. A number of mocking frameworks exist for various languages, of which Mockito [27] and Moq [28] are some of the most popular options for Java and C#, respectively. For our purposes, it is crucial that a mocked object behaves exactly like an instance of the corresponding type unless an instance field or function is specifically redefined by the user. DMock relies on both Mockito and Moq to support this functionality, which is sometimes also referred to as *spying*.

7 Conclusions

In this paper, we presented a toolkit for automated testing of Dafny programs: DUnit (unit testing framework), DMock (mocking framework), and DTest (automated test generation). The main component of the toolkit, DTest, works with the Boogie representation of a Dafny program to generate tests that (i) target branch coverage of the compiled code and (ii) contain runtime assertions extracted from method specifications in the Dafny code. We evaluated the coverage DTest achieves on several preexisting Dafny programs, showed that it can help identify unexpected behavior in external libraries, and compared it to an alternative more costly solution that optimally minimizes the number of tests. Overall, our results show that DUnit, DMock, and DTest are a promising toolkit for automatically generating high coverage tests for Dafny.

Acknowledgments The authors would like to thank Aleks Chakarov, Cody Roux, William Schultz, and Serdar Tasiran for their invaluable feedback on the usability of the toolkit, Ryan Emery and Tony Knapp for facilitating the use of the ESDK as a benchmark dataset, Rustan Leino, Mikael Mayer, Aaron Tomb, and Remy Williams for their feedback on the source code, and the anonymous reviewers for helping improve this text. This work is partly supported by an Amazon post-internship graduate research fellowship.

References

1. de Azevedo Oliveira, D., Medeiros, V., Déharbe, D., Musicante, M.A.: BTestBox: a Tool for Testing B Translators and Coverage of B Models. In: Tests and Proofs. pp. 83–92 (2019). https://doi.org/10.1007/978-3-030-31157-5_6
2. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Computing Surveys **51**(3) (2018). https://doi.org/10.1145/3182657
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387 (2005). https://doi.org/10.1007/11804192_17
4. Becker, B.F.H., Lourenço, C.B., Marché, C.: Explaining Counterexamples with Giant-Step Assertion Checking. In: Workshop on Formal Integrated Development Environment. pp. 82–88 (2021). https://doi.org/10.4204/EPTCS.338.10
5. Beyer, D., Chlipala, A., Henzinger, T., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: International Conference on Software Engineering. pp. 326–335 (2004). https://doi.org/10.1109/ICSE.2004.1317455
6. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative Verifier-Based Testing. In: Fundamental Approaches to Software Engineering. pp. 389–408 (2019). https://doi.org/10.1007/978-3-030-16722-6_23
7. Boogie, https://github.com/boogie-org/boogie
8. Chakarov, A., Fedchin, A., Rakamarić, Z., Rungta, N.: Better Counterexamples for Dafny. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 404–411 (2022). https://doi.org/10.1007/978-3-030-99524-9_23
9. Christakis, M., Leino, K.R.M., Müller, P., Wüstholz, V.: Integrated Environment for Diagnosing Verification Errors. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 424–441 (2016). https://doi.org/10.1007/978-3-662-49674-9_25
10. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: International Conference on Functional Programming. pp. 268–279 (2000). https://doi.org/10.1145/351240.351266
11. Coverlet, https://github.com/coverlet-coverage/coverlet
12. Dafny, https://github.com/dafny-lang/dafny
13. AWS Encryption SDK, https://github.com/aws/aws-encryption-sdk-dafny
14. Filliâtre, J.C., Paskevich, A.: Why3—Where Programs Meet Provers. In: European Symposium on Programming. pp. 125–128 (2013). https://doi.org/10.1007/978-3-642-37036-6_8
15. Hao, D., Zhang, L., Wu, X., Mei, H., Rothermel, G.: On-Demand Test Suite Reduction. In: International Conference on Software Engineering. pp. 738–748 (2012). https://doi.org/10.1109/ICSE.2012.6227144
16. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad Apps: End-to-End Security via Automated Full-System Verification. In: Symposium on Operating Systems Design and Implementation. pp. 165–181 (2014)
17. Hsu, H.Y., Orso, A.: MINTS: A General Framework and Tool for Supporting Test-suite Minimization. In: International Conference on Software Engineering. pp. 419–429 (2009). https://doi.org/10.1109/ICSE.2009.5070541

18. Irfan, A., Porncharoenwase, S., Rakamarić, Z., Rungta, N., Torlak, E.: Testing Dafny (Experience Paper). In: International Symposium on Software Testing and Analysis. pp. 556–567 (2022). https://doi.org/10.1145/3533767.3534382
19. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 553–568 (2003). https://doi.org/10.1007/3-540-36577-X_40
20. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: World Congress on Formal Methods. pp. 806–809 (2009). https://doi.org/10.1007/978-3-642-05089-3_51
21. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370 (2010). https://doi.org/10.1007/978-3-642-17511-4_20
22. Leino, K.R.M.: Accessible Software Verification with Dafny. IEEE Software **34**(6), 94–97 (2017). https://doi.org/10.1109/MS.2017.4121212
23. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4
24. Dafny utilities library, https://github.com/dafny-lang/libraries
25. Liew, D., Cadar, C., Donaldson, A.F.: Symbooglix: A Symbolic Execution Engine for Boogie Programs. In: International Conference on Software Testing, Verification and Validation. pp. 45–56 (2016). https://doi.org/10.1109/ICST.2016.11
26. Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2019). https://doi.org/10.1109/TSE.2019.2946563
27. Mockito, https://github.com/mockito/mockito
28. Moq, https://github.com/moq/moq
29. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
30. Polikarpova, N., Furia, C.A., West, S.: To Run What No One Has Run Before: Executing an Intermediate Verification Language. In: International Conference on Runtime Verification. pp. 251–268 (2013). https://doi.org/10.1007/978-3-642-40787-1_15
31. Spettel, P.: Delfy: Dynamic Test Generation for Dafny. Master's thesis, Eidgenössische Technische Hochschule Zürich (2013). https://doi.org/10.3929/ethz-a-010056933
32. XUnit, https://github.com/xunit/xunit
33. Z3, https://github.com/Z3Prover/z3